Motivation: cloud services that span the globe has made it easy to deploy geographically distributed systems that span continents and oceans. This provides the opportunity for:

- Better performance by preventing bottlenecks and decreasing latency (localizing accesses).
- Durability to catastrophic failure (the loss of an entire data center)
- Serve larger communities and international audiences

Problem: latency between data centers and the high likelihood of network partitions makes strong consistency a challenge, however, eventual consistency is not acceptable **Solution:** allow as much of the system as possible to operate as independently as possible by separating placement decisions from access decisions (e.g. a root quorum and subquorums).

Problem: centralized systems administration is impossible in a global context. **Solution**: the root quorum must be able to adapt the system automatically by observing conditions and access patterns and applying policy-driven changes to the system in real-time.

Problem: microservice architectures that separate concerns (e.g. lock servers, access servers, migration servers, distributed file systems, true-time) create very complex consistency conditions that are impossible to reason about globally. **Solution:** there must be an intersection between the root quorum and subquorums -- this also improves the global availability of the system as all nodes participate in both consensus decisions.

Problem: consensus algorithms do not scale well and the root quorum requires global participation. **Solution**: delegated voting ensures that a smaller number of more responsive nodes can participate safely in root quorum decisions.

Problem: different objects have different access patterns and different applications have different requirements for availability, durability, etc. **Solution**: allow subquorums to implement different consensus protocols (ePaxos, Raft) and to have different geographic placement patterns.

Problem: objects are placed as close as possible to where they are accessed most, ensuring that objects in the same subquorum are related and allowing for local transactions; however less frequent accesses or transactions might cross subquorum boundaries. **Solution**: remote reads and writes ensure that client-side transactions can be created and committed across multiple subquorums.

Problem: access patterns can change over time and object relationships shift. **Solution:** epoch changes migrate objects and subquourms to optimize current network conditions and access patterns.

Hypothesis: hierarchical consensus is a framework for providing flexible strong consistency that can scale to hundreds of geographically distributed nodes that dynamically adapt to changing conditions improving both system performance and maintainability.

Introduction

An Architecture for Flexible Global Consensus (Background)

Regionalization of node placement (for most Cloud providers):

- Rack: lowest latency but most susceptible to group failure
- Zone: within the same data center, no wide-area links
- District: across data centers that may be up to 50 miles apart
- Region: across districts that provide “user acceptable latency” e.g. central europe or us west coast
- Planetary: across regions, continents, or oceans

Figure: an architectural overview of root quorum in all regions and subquorums serving different regions.

Geo-Distributed Consensus

Raft: leader-oriented consensus requires one round of communication but is unavailable when the leader fails. Well suited to high availability primary backup in a single region.

Mencius: round-robin leadership is best suited to balancing requests between two regions, particularly if requests are balanced more toward one region than another. Two rounds of communication are required in the worst case.

ePaxos: optimistically allows for one round of communication but requires three in the case of conflicts, best suited to distributing quorum nodes each in their own region.

Figure: consensus does not scale

The issue is that

Access Patterns and API

Systems are extremely sensitive to access patterns though most applications and systems are only optimized for one type of access pattern. Real-world systems have multiple types of access patterns including:

Figure: Placement of objects in different regions and quorum configurations based on access patterns.

Single ownership: objects are primarily accessed in one region and do not migrate.

Revolving access: object accesses migrate through space and time, e.g. objects are more frequently accessed during daylight hours.

Conflicting access: objects are continuously accessed from multiple regions without transitions.

A Framework for Adaptive Systems

Requirement 1: Systems should be as fluid as the information they contain

Requirement 2: No partial failures

Requirement 3: Consistency semantics must be transparent and interpretable

Hierarchical Consensus

Figure: hierarchical consensus with delegated voting

Delegated Voting

Epoch Transitions

Elastic Membership

Adaptability

The system is designed to respond to changes in access patterns and network conditions.

Liveness and Access Monitoring

Liveness - nodes send heartbeats to their delegates (even in non-leader based subquorums) and delegates report heartbeats to the root node (e.g. Niobe and vertical Paxos).

Accesses - subquorums are responsible for tracking access counts on a per-object/per-region basis and periodically forwarding access probabilities to the root node.

Neither liveness monitoring nor access monitoring need be preserved across epochs.

Object Placement and Migration

Data-Driven Policies

Epoch Operations

Remote Accesses

Transactions

Safety

Expected Failure

Assassination

Nuclear Option

System Implementation

A single process implementation - one node, one process.

Implemented in Go with the SEDA paradigm/Actor model to enhance concurrency of message passing between processes and concerns within a single process while ensuring a single linearization of operations between all subquorums and root quorum operations handled by the process.

Go-routines ensure that a single node's computational resources are fully utilized, even in a containerized context. Containers add more nodes, not more fragmentation.

Event-Driven Design

Optimizations

- Receive-side thriftiness
- Message aggregation
- Prefer in-memory commit without waiting for disk (repair on demand)
- Bootstrapping subquorums and root quorums

Experimental Evaluation

Discussion

Consistency Semantics

Exporting a global log (there exists only a single ordering of events).

Related Work

Engineering solutions

Conclusion